JUL 26 2004

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** 22.Jul.04	**3. REPORT TYPE AND DATES COVERED** MAJOR REPORT
4. TITLE AND SUBTITLE FAULT TOLERANCE IN NETWORKED CONTROL SYSTEMS THROUGH REAL-TIME RESTARTS	**5. FUNDING NUMBERS**	
6. AUTHOR(S) MAJ GRAHAM SCOTT R		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) UNIVERSITY OF ILLINOIS AT URBANA	**8. PERFORMING ORGANIZATION REPORT NUMBER** CI04-491	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) THE DEPARTMENT OF THE AIR FORCE AFIT/CIA, BLDG 125 2950 P STREET WPAFB OH 45433	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES		
12a. DISTRIBUTION AVAILABILITY STATEMENT Unlimited distribution In Accordance With AFI 35-205/AFIT Sup 1	**12b. DISTRIBUTION CODE**	

13. ABSTRACT *(Maximum 200 words)*

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

20040809 070

14. SUBJECT TERMS			**15. NUMBER OF PAGES** 21
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**

BEST AVAILABLE COPY

Networked control consists of sensors and actuators that interact with the "real world" environment (*plant*), which are connected to each other and to with controllers over digital networks. Such systems are long-running, have real-time requirements, and should function in a correct and timely manner even in the presence of failures and software upgrades. Arguably, the growth and the widespread use of the internet, the largest operational distributed system today, has been enabled by its fault-tolerant properties such as robustness to router and link failures [20]. As such systems take on an increasingly critical role, supporting local and national infrastructures, homes and industries, the importance of reliability and availability issues grows.

There are multiple causes of individual component failure in a networked control system including software bugs and transient errors. The latter class of errors has been shown to be responsible for a large number of residual faults in individual components [16, 19]. In practice, a large number of the aforementioned failures result in crash failures, resulting in the component being restarted. In addition to unplanned restarts, planned software upgrades to components also result in restarts. The importance of fast individual component restarts (Microreboots) to reduce mean-time-to-recovery (MTTR) and thus improve system availability has been discussed in [14, 5].

In networked systems, fault-tolerance refers to maintaining correct and timely system operation by tolerating errors in individual components, instead of avoiding failure of each individual component. This notion of fault-tolerance in distributed real-time control systems has been described in [9]. Generic fault-tolerance techniques developed in transaction based systems [22] and scientific computing [11, 1] are not applicable in this domain due to their high overhead and lack of real-time performance. The goal of this work is to tolerate restarts and ensure correct and timely operation of a networked control system in the presence of restarts to individual components. Achieving this goal increases the mean-time-to-failure (MTTF) of the system thus enhancing *both the reliability and the availability* of the system.

In the IT Convergence Laboratory at the University of Illinois, we have developed a traffic control system testbed as prototype networked control system. The testbed consists of a set of cars operating in an indoor track. Each car is independently controlled by its own software controller that achieves to higher level goals and receives feedback about its position from a vision system (sensor). The car controller then sends control updates to the car. This system is described in further detail in section 2.

Through our experience with the traffic control testbed we identified three key challenges towards achieving our goal: (1) Satisfying the real-time requirements on the control updates sent to the controlled plant, (2) Ensuring reliable delivery of aperiodic commands or messages between components, and (3) Ensuring that components can restart independently without cascading the restart to another component.

Specifically, this work focuses on the first challenge above, that of guaranteeing timely control updates in the presence of restarts to the sensors and the controllers (Actuators are typically simple and more robust). Our approach exploits two important domain-specific features of the system in order to address this challenge, the presence of a *state estimator* and the computation of a sequence of future controls by the controller that are buffered at a *control buffer* in the actuator. We use these properties of the controller and its recovery times to derive an upper bound on the inter-arrival time between sensor updates. We have developed a *control model based checkpointing* technique where we analyze the properties of a controller and derive the condition under which the controller state needs to be regenerated from a checkpoint upon restart. Both the above analyses are offline and are performed for each controller locally, thus making them highly scalable. The second and third challenges above are integral to our goal. We address them in the context of our testbed. We describe an approach to guarantee reliable delivery of aperiodic messages and to ensure that the component state due to these messages is regenerated upon recovery. Our design of the components in the testbed avoids cascading restarts due to the restart of a single component and thus eliminates *restart dependencies*.Component A is said to be *restart dependent* on component B if restarting component B entails the restart of component A. In a networked control system such as our testbed, restart dependencies are a result of the nature of the communication channels between components. Throughout this paper, we focus on the restart of the controller in a system following the sensor-controller-actuator pattern. The controller is assumed to operate in a single mode.

Our example scenario in this paper consists of one car tailgating another. The two cars operate autonomously, i.e. they are controlled by independent controllers. We evaluate the effectiveness of the techniques mentioned above by measuring the deviation of the front car from its trajectory upon random restarts of its car controller. If the front car deviates by more than the separation distance, a collision results. This scenario captures all the challenges described above. Our original design of the components in this system was not restart-tolerant due to the presence of restart dependencies and the potential violation of the real-time deadlines of the control updates due to inability to re-establish state upon restart of a controller.

The main contributions of this work are the following:

1. An offline analysis technique that derives the condition under which real-time deadlines of the system are satisfied without checkpointing the controller state periodically.

2. Experimentally validating the above analysis technique for real-time control updates, re-engineering controllers and sensors to remove restart dependencies and ensuring reliable delivery of aperiodic messages in the presence of restarts in our traffic control testbed.

The rest of the paper is organized as follows. Our traffic control testbed is described in section 2. Section 3 formally defines the real-time requirements of a controller and describes the issues related to satisfying these requirements in the presence of restarts. Section 4 describes the techniques applied to eliminate restart dependencies between components and to ensure the reliable delivery of aperiodic commands. Our experiences in applying our techniques to the traffic control testbed are evaluated in section 5. We describe existing work in section 6 and conclude with section 7.

2 The Traffic Control Testbed

We have developed a traffic control testbed as a research prototype of a networked control system [15]. An overview of the system [2] is presented, followed by some design enhancements to tolerate variations in operating conditions. These enhancements are used to develop restart techniques in later sections.

2.1 Traffic Control Testbed

Fig. 1 illustrates our traffic control testbed which consists of a fleet of small cars running on a small indoor track. Each car is controlled by a dedicated controller, running on a laptop computer. This controller periodically computes a series of controls and sends them to an actuator. The actuator component sends controls to a car via the serial port, a micro-controller, a dedicated RF transmitter, and a dedicated RF

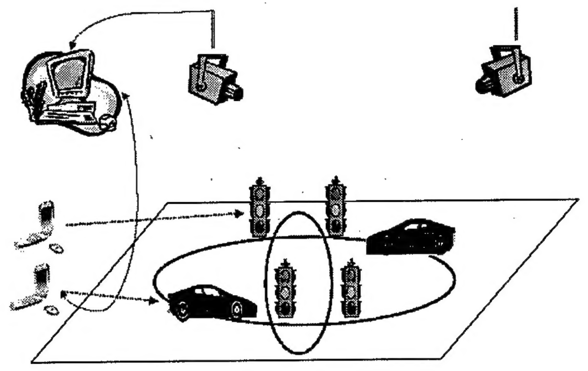


Figure 1: A Wireless Control Network Testbed

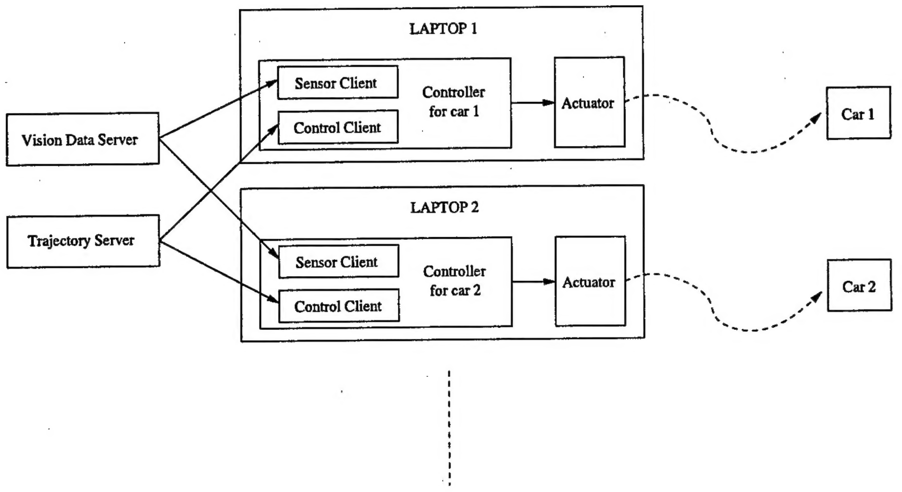
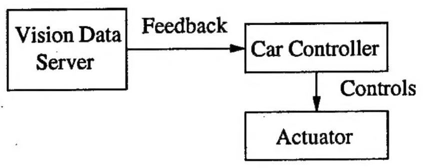


Figure 2: Implementation of Testbed

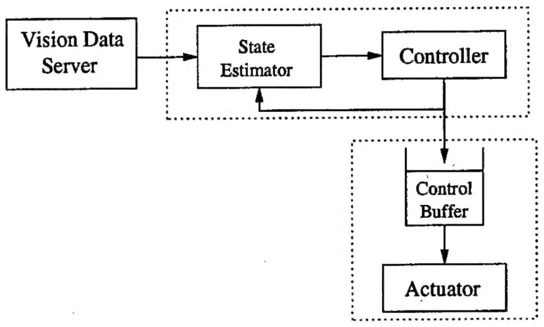
receiver. The received signals are used to control motors on the physical car to change its speed and direction. For our purposes, this configuration is equivalent to a controller being physically on the car

The cars are tracked by a vision system consisting of two ceiling mounted cameras and two vision servers. The vision servers extract position and orientation information for each car, then relay the information to a data server which in turn disseminates this data to the controllers. The vision server along with the data server functions as a sensor. Communication between the vision system, the data server and the controllers is via wired or ad hoc wireless networks.

Figure 2 shows the software architecture of the controllers driving each car. To begin an experiment, a controller receives a trajectory to follow from a trajectory server. Throughout the experiment, a controller



(a) Original design



(b) Enhanced design

Figure 3: Design enhancements for robustness

receives periodic updates from the vision servers. The controller periodically computes a set of controls according to the vision updates, the desired trajectory, and the current time and sends it to the actuator.

2.2 Design for Robustness

In most networks, packet delivery is usually performed on a best effort basis. On wireless networks in particular, packet delays are unpredictable due to interference and noise. Hence, design enhancements are required to control the plant despite such variations.

Figure 3(a) describes a control loop in the testbed design of Figure 2. In the testbed, feedback from the vision data server to the controller is transmitted over a network, and is subject to unpredictable delays. For robust design and simpler deterministic control, the controller needs to be shielded from these delays. Hence, as shown in figure 3(b), the controller is isolated from the vision data server by a *state estimator*. The estimator estimates the state of the car as a function of the feedback and controls sent to the actuator. E.g. A calibration of the steering angles and the speed of the car for different control outputs is used in our testbed as a state estimator. Based on these estimates, the controller can operate even when some feedback updates are delayed or lost. Similarly, a *control buffer* isolates the actuator from delays and losses of controls from the controller. To exploit this, the controller computes a sequence of future controls, which is then stored in the control buffer. Hence, in case of delays in the arrival of future controls, the actuator can still operate using the buffered controls.

While the state estimator estimates the position and orientation of the car, the car controller itself uses model predictive control [3], and computes possible future sequences of controls based on the current state estimate. After each control, the state of the car is predicted using a *state predictor*, which uses techniques similar to the estimator. The controller then selects the sequence of controls that results in the least squared error between the predicted state of the car and the desired trajectory, and sends this to the actuator.

The above design enhancements are necessary for timely system operation. We can exploit these enhancements to improve system robustness in the presence of restarts. For instance, the state estimator allows the controller to tolerate longer delays in sensor updates. This facilitates fault management techniques such as extending the deadline by which the vision data server can be restarted. To ensure that the system can operate despite such faults, the deadlines need to be determined and efficient restart mechanisms developed. This is addressed in the following section.

3 Real-time Analysis of Controller Restarts

This section presents an analysis of the enhanced design of the control system from section 2.2. Constraints on maximum delay of sensor updates are derived and used to select appropriate restart techniques for the controller.

3.1 Controller model

We now model a generic controller. This will be used to derive various bounds in the rest of the section. A controller may be modeled as the tuple $(I_p(t), I_a(t), x(t), u_p(t), u_a(t))$, at a given time t, where:

- I_p is the vector of periodic inputs. For example, the car controller of Figure 1 receives periodic feedback updates from the Vision Data Server.
- I_a is the vector of aperiodic inputs. For example, the car controller may receive aperiodic inputs about an approaching car that may collide with it.
- x is the state of the controller. For a car controller, this is the position, speed, and steering angle of the car.

- u_p is the vector of periodic control outputs. For example, the car controller sends periodic controls to the Actuator.

- u_a is the vector of aperiodic control outputs. For example, the car controller gets aperiodic controls (STOP/GO) from the Trajectory Server.

In the following analysis, we only model the steady state operation of a controller such as the car controller. We consider only periodic inputs I_p and assume that the controller only generates periodic outputs u_p. Aperiodic controls and updates are considered in a later section.

3.1.1 State Estimator

As illustrated in Section 2.2, noise and perturbations in periodic inputs are tolerated using a state estimator. We use a Kalman filter [7] to estimate state in the car controller. This is a commonly used state estimator.

An extended Kalman filter maintains two state estimates $\tilde{x}$ and $\hat{x}$, which are updated using the following difference equations:

$$\tilde{x}(t) = f(\tilde{x}(t-1), u_p(t-1), w(t-1)) \tag{1}$$

$$\hat{x}(t) = h(\tilde{x}(t), I_p(t), v(j)) \tag{2}$$

where the random variables w and v represent the process and measurement noise respectively. Eq. 1 updates the estimate based on the previous control, while eq. 2 corrects the estimate using current observations I_p. The estimate $\hat{x}$ is the output of the filter and is used to compute controls.

In the rest of the section, for simplicity, we assume that I_p is a scalar. However, the following analysis can easily be extended to the case where I_p is a vector.

As noted in Section 2.2, packet delays and sensor restarts may cause successive updates of I_p to be delayed or lost. Since discrete control is imperfect, estimates are noisy due to process noise w, and corrections are not incorporated by eq. 2, the error between the estimate $\tilde{x}$ and the actual state x increases with time.

Figure 4 illustrates the increase in uncertainty of state estimates, when successive updates I_p are lost. During this period, the controller can only use the uncorrected estimate $\tilde{x}$ to compute controls. $SE(k-1)$

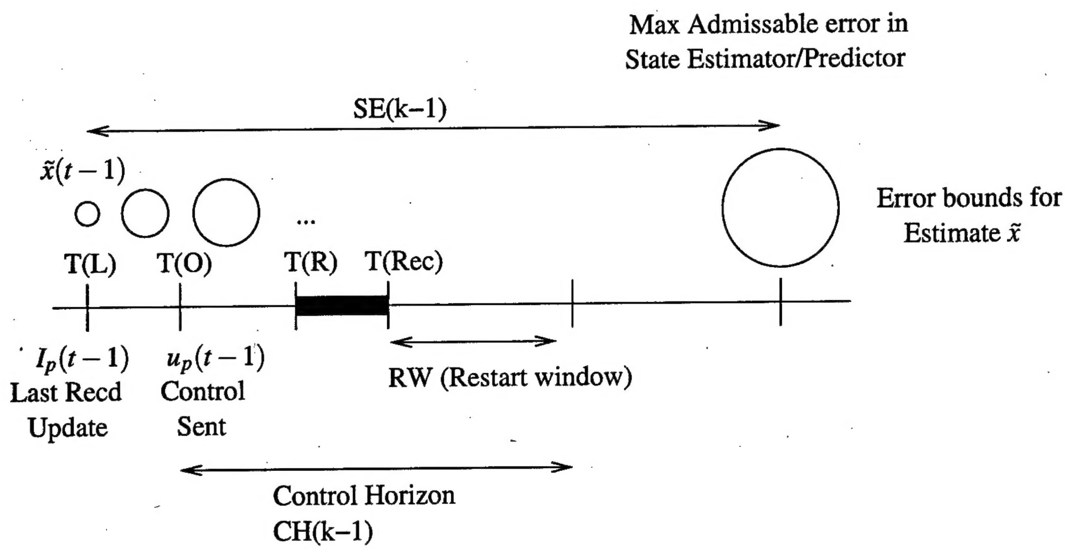

Max Admissable error in State Estimator/Predictor
SE(k−1)
$\tilde{x}(t-1)$
...
Error bounds for Estimate $\tilde{x}$
T(L)
T(O)
T(R)
T(Rec)
$I_p(t-1)$ Last Recd Update
$u_p(t-1)$ Control Sent
RW (Restart window)
Control Horizon CH(k−1)

Figure 4: Time line illustrating estimate and control buffer deadlines

represents the interval, after the last update $T(L)$ was received, up to which the error in estimate is tolerable. For proper controller operation, the maximum delay between successive sensor updates $T_{u_{max}}$ should be less than this interval, i.e.,

$$T_{u_{max}} < SE(k-1) - t_u - T(proc) \tag{3}$$

where t_u is the control output period, and $T(proc)$ is the processing time at the controller. Note that equation 3 can also be used to derive the deadline for sensor restarts in terms of $T_{iu_{m}ax}$.

3.1.2 Control Buffers

As noted in Section 2.2, a control buffer is used to tolerate delays and losses of controls sent over a network. The interval of time for which the Actuator has buffered future controls is called the *Control Horizon*. Since the state estimate is used to compute the controls, the control horizon should be lesser than the interval determining tolerate error in state estimates, i.e.,

$$CH(k) < SE(k) \tag{4}$$

The control horizon can also be limited by system limitations such as the computational resources. For example, a model predictive controller may analyze only a limited horizon of future controls, which is then sent to the actuator. For safety, the last control in the sequence can be a fail-safe.

3.2 Restart analysis and techniques

We now analyze controller restart bounds, based on the model presented in Section 3.1. This will motivate techniques to restart the controller and restore its state. In the following analysis, we assume that restarts to the controller are infrequent compared to the frequency of the sensor inputs or the controls sent to the actuator. Specifically, we assume that the interval between restarts is greater than the period of successive lost updates. This is usually a reasonable assumption in practice.

Figure 4 shows the time line for a controller restart. The controller has a fault and restarts at time $T(R)$. It then recovers and completes all initializations at time $T(Rec)$. The thick line in the figure represents this restart period of the controller. $T(O)$ is the time the last output was sent by the controller before the restart occurred. By the description of Section 3.1, the controls sent to the actuator are valid until time $T(O)+CH(k-1)$. Hence, the controller needs to send the next controls only before this deadline. The time interval from $T(Rec)$ to $T(O)+CH(k-1)$ is called the *restart window* (RW) of the controller. In the worst case, RW is given by the following equation:

$$RW = CH(k-1) - (T(Rec) - T(R)) - t_u \tag{5}$$

Upon recovery, the controller needs to send a control output to the actuator (consisting of a sequence of future controls) before the restart window expires.

The state x in the controller has to be re-established, upon restart and recovery, for controls to be computed before the deadline RW above. This allows the controller to resume normal operation without affecting the Actuator. We consider two different ways to re-establish controller state:

1. **State check-pointing:** Upon sending each output, the controller checkpoints the car state and the sequence of controls sent to the actuator. Upon restart-recovery, this checkpoint is read and the state at the time of recovery is computed by a state estimator. In the case of the Kalman state estimator, at time $T(0)$ in figure 4, we need to checkpoint the state $x(k-1)$ measured at the last sensor update received at time $T(L)$, and the controls sent since the measured update.

2. **No state check-pointing:** The state of the controller can be re-established before time $T(O)+CH(k-$

1), if a sensor updated is received within the restart window. This avoids overhead due to check-pointing. However, this condition imposes a tighter upper bound on $T_{u_{max}}$ than equation 3. This bound is given by:

$$T_{u_{max}} < RW - T(Proc) \qquad \Longrightarrow \qquad T_{u_{max}} < CH(k-1) - (T(Rec) - T(R)) - t_u - T(Proc) \quad (6)$$

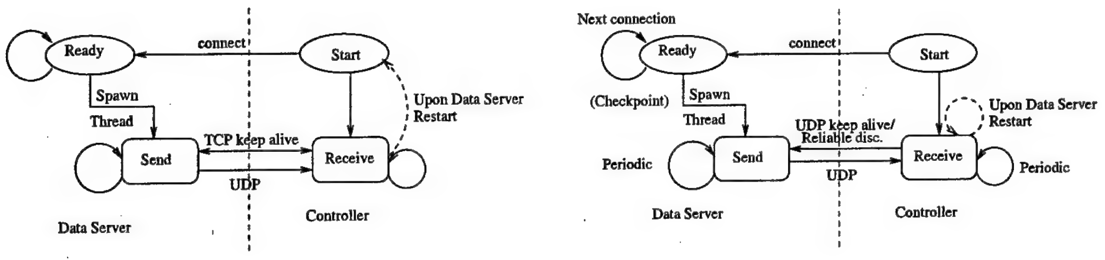
Equation 6 is the condition under which state x can be regenerated upon recovery *without periodically check-pointing* x and the set of future controls. This equation subsumes the bounds in equations 3 and 4.

In practice, we can calculate the size of the restart window of the controller using the length of the control horizon, the worst-case recovery time and the period of the control output using equation 5 above. We then use equation 6 to calculate the maximum inter-arrival time of the sensor updates in order that the controller state is re-established upon restart without periodic check-pointing of the state. If this condition holds for the given sensor, then check-pointing state periodically in the controller is not necessary. Otherwise, check-pointing is required. We thus refer to this technique as *control model based check-pointing* of periodic state. Note that the constraint on the inter-arrival time of the sensor updates $T_{u_{max}}$ imposes requirements on both the sensor as well as the communication medium between the sensor and the controller. This bound needs to be satisfied even in the presence of restarts to the sensor. We have experimentally validated the mechanism developed in this section by applying them to the car controller in our testbed as described in section 5.

In the testbed, in addition to the position of the car, the speed of the controlled car before restart needs to be regenerated upon recovery, due to the inertial property of the car. In our experiment, speed is a state variable that changes infrequently and aperiodically. We discuss regeneration of component states corresponding to aperiodic updates sent to the controller in the following section.

4 Restart Techniques: Dependencies and Aperiodic Updates

Apart from maintaining the real-time properties of the system in the presence of restarts to individual components, we identified two other significant challenges that needed to be overcome in order to maintain correct system operation in the presence of restarts to a single component: eliminating restart dependencies



(a) Original design (b) Enhanced design

Figure 5: Removing communication-based restart dependencies

and regenerating the component state due to aperiodic updates upon recovery. Below, we discuss these two challenges in the context of our testbed.

4.1 Restart Dependencies

A component A is restart dependent on component B, if restarting component B entails restarting component A. We observe two sources of restart dependencies in networked control systems such as our testbed: (1) inability to independently restart each component and (2) communication-based restart dependencies. Both these sources of dependencies are illustrated in section 4.1.1 The messages exchanged between components are either periodic (e.g. sensor updates, control outputs) or aperiodic (e.g. trajectories, STOP/GO). Thus data dependencies that arise from these messages are addressed by our techniques section 3 and 4.2 respectively.

4.1.1 Restart Dependencies in the Testbed

Communication-based Restart Dependencies

Our original design of the vision data server, the trajectory server, and the controller contained communication-based restart dependencies between the controller and the data server as well as the controller and the trajectory server. The design however did not contain any communication-based deadlocks. The components needed to be redesigned in order to avoid cascading restarts in the event of restarting one of the components. We use the communication between the data server and the controller to illustrate our design.

Fig 5(a) represents the communication between the data server and a particular car controller in the original design. The car controller receives updates from the corresponding thread in the data server via a UDP connection. A TCP connection between the car controller and the data server is used as a keep-alive for the data server to detect a restart of the car controller and 'garbage-collect' the corresponding thread in the data server. In this design, restarting the data server caused the TCP connection to disconnect, thus resulting in an error in the car controller which thus needs to be restarted (represented by the dashed arc). This is an instance of a restart dependency. Fig 5(b) represents our new design of the communication between the data server and the controller which eliminates this dependency. In this implementation, the TCP connection used as a keep-alive between the data server and the controller is eliminated. The data server now checkpoints its connections, thus re-establishing this state in the event of a restart recovery. The controller uses UDP keep-alives or reliable disconnection messages in order that the data server can garbage-collect its corresponding thread. Thus, in the event of restarting the data server, the controller continues to wait for an update from the data server and does not need to be restarted. We apply a similar technique to re-design the communication between the controller and the trajectory server.

Restart Independence

The communication between the controller and the actuator itself uses UDP over a reliable link and thus does not give rise to any restart dependencies. In the original design, the controller and the actuator were a part of the same component. The controller sent the control signals via serial port to the micro-controller which, in turn, sent the signals via a transceiver to the receiver on the car which drives the motors on the car. While the actuator is required to be a highly reliable component, the complexity of the controller precluded this. Inability to restart the controller independently of the actuator made the system intolerant to controller restarts. Our new design split the controller and the actuator functionality allowing the two software components to communicate via a reliable UDP link, making the actuator simple and robust and the controller and actuator independently restartable.

4.2 Aperiodic Updates

Aperiodic updates cause discrete changes to the component state corresponding to these updates (*aperiodic component state*) in the component. These updates are critical and need to be delivered reliably. In our domain, we encounter the following categories of aperiodic updates: (1) sensors to controllers - e.g. an aperiodic event such as a gunshot detected by the motes and sent to the car controller, (2) controllers to a sensor - e.g. a controller registering with (or disconnecting from) a sensor for updates upon start-up, (3) higher level controller to a lower level controller - e.g. a trajectory or a *GO/STOP* command sent by the trajectory server to the controller, and (4) lower level controller to a higher level controller - e.g. a controller registering with a trajectory server upon start-up.

In order to handle aperiodic updates in the event of restarts to the component, (a) the aperiodic component state needs to be regenerated upon restart and (b) aperiodic updates need to be delivered reliably. We illustrate these two mechanisms with respect to the car controller in our testbed.

In the testbed, the car controller needs to regenerate the *GO/STOP* status, trajectory, and the speed of the car. Our approach involves checkpointing the aperiodic component state each time an aperiodic update is received or the aperiodic component state changes. This checkpoint is read and used upon recovery. Also, each of these aperiodic updates need to be delivered reliably to the car controller. These aperiodic updates are thus sent using an application-level protocol where the updates are re-sent until an acknowledgment is received from the controller. Acknowledgments are only sent by the car controller after the aperiodic component state change due to the update is checkpointed at the controller. If the controller is down when an aperiodic update is sent, this update is received by the controller after recovery.

5 Results

Our testbed described in section 2 has been designed to explore issues in the convergence of control with communication and computation. It has been tested for different scenarios including leader-follower mode and a set of independent cars achieving individual goals without collisions. Videos of these demonstrations can be viewed at `http://decision.csl.uiuc.edu/~testbed/Videos.html`. Restarting controllers and sensors is common during experimentation and bug fixes. In this section, we evaluate our techniques from section 3 and 4 using a tailgating experiment.

5.1 Case Study: Tailgating

To study the effect of controller restarts, we derive an experiment from the scenario of a car closely following another car along an identical trajectory with a fixed separation of 100mm 'bumper-to-bumper' along an elliptical trajectory which is 9.6m long. The cars themselves are 225mm long and travel at an average speed of 371mm per second. One loop around the oval takes 26s to complete. The cars are controlled independently. The car in front is subject to real-time constraints in the presence of restarts to its controller since its maximum deviation is bounded by its distance from the car behind. The experiment consisted of sporadically restarting the controller of the first car. This experiment was repeated 15 times. Below, we show the results for a representative run. The goal of our restart techniques in this application is to avoid collisions.

Although the experiment is derived from the case of one car following another, we are primarily interested in the deviation of the first car from its desired trajectory in the event of restarts. The plot of the deviation of the car from its trajectory demonstrates the physical manifestation of a restart and its temporal characteristics while using our restart techniques. A restarted car may exhibit a temporal deviation from its desired position. For the purposes of capturing data, we measure the difference in the position of the car from the position of a fictitious car exactly following the desired trajectory. Note that the inherent noise in the measurement and the actuators causes the car to deviate from its trajectory by approximately 25mm even under normal operation.

The control horizon of the future controls sent by our car controller to the is 1800ms. The period of sensor updates to the controller and the controls sent to the actuator is 100ms. The restart recovery time is the time between killing the controller and the re-establishment of its communication channels after recovery. This time is typically 500ms. Note however, that in our design of the controller, the controller connects to the data server upon restart and waits for a response, upon whose receipt the controller is said to be completely recovered. This is, in fact, a residual dependency in our system that can cause much higher restart recovery times for the controller if the data server delays the response. This however, does not occur in practice since we do not restart the controller and the data server at the same time. By our analysis in

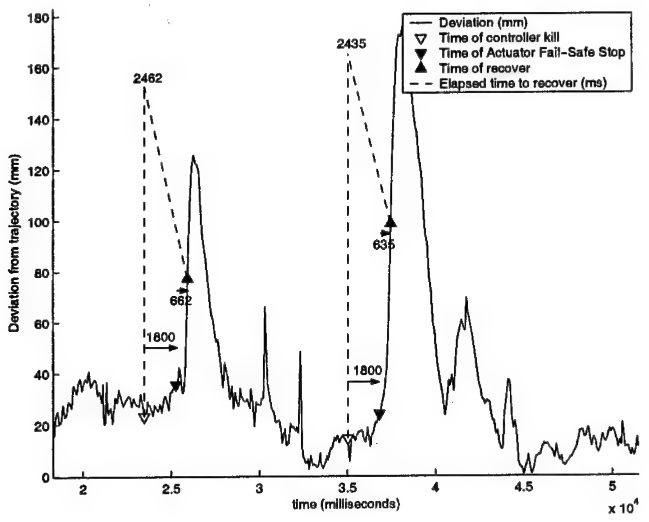


Figure 6: Unsuccessful restarts

section 3, the restart window in our case is (1800 - 500 - 100) = 1200ms, which is greater than the typical sensor update period of 100ms and the worst case sensor update inter-arrival time that occurs in the event of a sensor restart. Thus, we *do not checkpoint the periodic state* in the controller. Upon recovery, the controller simply waits for the following sensor update in order to compute and send controls to the actuator.

Since a successful result in this case shows no deviations from the desired trajectory, we illustrate our techniques by introducing delays in the restart recovery times in order to compare unsuccessful restarts with successful restarts. Figure 6 shows two such large deviations (2462 and 2465ms respectively). Note that the elapsed time for restarting, including the forced delay, exceeds the restart window. Hence, in both restarts, the actuator stops the car after 1800ms (due to its fail-safe mechanism), after which the deviation from the desired trajectory begins to grow quickly. Also, note the slower growth in deviation prior to the time at which the actuator stops the car. This is a result of graceful degradation of control commands. Upon recovery, the car gains speed and catches up with its desired trajectory and the deviation stabilizes. Note that in these cases, the restart windows are non-existent due to the high recovery times.

Figure 7 plots a the effect of sporadic restarts of the car controller with randomized recovery times. The car is allowed to stabilize and operate normally between restarts. Actuator noise is frequently larger than the effect of a restart. Therefore, we have chosen a trajectory in which actuator noise is minimized. Specifically,

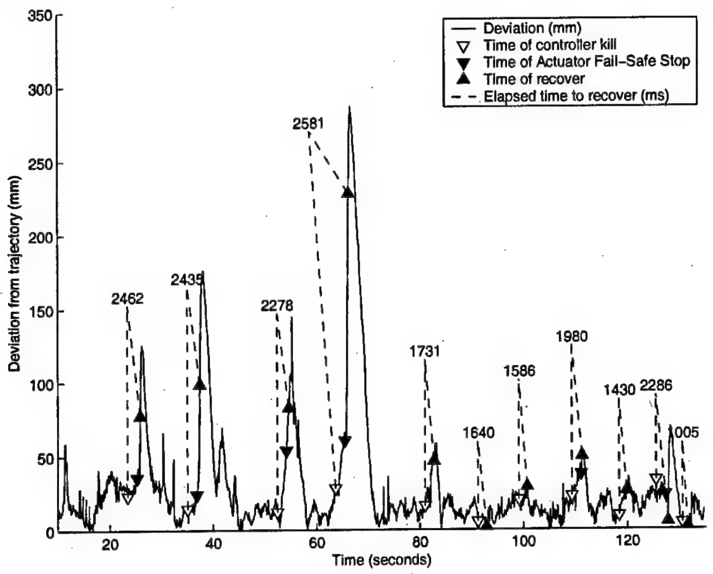


Figure 7: Sequential restarts

we use a very large oval in order to eliminate noisy sharp turns as trajectory and command a constant speed. These choices are useful for clean illustration and are not required by our techniques.

We begin the experiment by running the system normally in order to demonstrate the typical actuator and sensor noise. A successful restart does not affect the car and the deviation from the trajectory is minimal. These restarts are characterized by the fact that the controller recovers before the entire set of future controls is used by the actuator, thus sending it a fresh sequence of controls i.e. the controller receives a sensor update within the restart window. In figure 7, the restart times are shown in milliseconds after each restart. The time of restart is represented by the inverted unshaded triangle and the time of recovery is represented by the upright shaded triangle. Restart times include both the recovery time as well as our randomly introduced delay. Inverted shaded triangles in the figure represent the time at which the control updates in the actuators are exhausted and the car stops due to the actuator fail-safe command. These have been plotted by simply adding the value of the control horizon (1800ms) to the time of restart. Successful restarts are those where there is no inverted shaded triangle. Clearly, restart recovery times above 1600ms lead to a restart window of 100ms or smaller (since the control horizon = 1800ms and the period of control updates = 100ms) causing the real-time deadline of the system to be violated. For restart recovery times above 1600ms and below 1700ms, regular checkpointing of controller state is necessary in order that the system deadlines are met.

The experiment demonstrates that all restarts which are accomplished within the prescribed safety horizon result in little or no deviation from the desired trajectory. Clearly, the control horizon and the tolerable error are application specific parameters.

6 Related Work

Restarts of individual components for recovery, pro-actively as well as reactively have been seen in software rejuvenation [25] and recovery-oriented computing [13, 4]. While the latter used fast restarts of individual components in a system as a tool to improve system availability, our work assumes a recovery time and maintains timely operation in the presence of restarts thus increasing the MTTF of the system aiding both reliability and availability. Also, these works are aimed at generic applications and do not address the real-time constraints that exist in control systems. Reducing the recovery time increases the size of the restart window and thus complements this work well.

Failure model in control systems is better described by the fail-bounded model [18]. Fail-bounded models ensure that wrong outputs have a bounded deviation from the correct output. Our use of the state estimator in the controller and the control buffer in the actuator assume this model. Fault-tolerance in distributed control systems have relied on redundancy and replication [12, 24, 6]. While replication is an effective technique against crash errors due to transient faults, it is expensive. Simple replication is ineffective against programmer bugs that cause all replicas to crash. Simplex [23] is an architecture that uses analytical redundancy to provide robustness against controller failure by switching to a back-up controller in real-time. Simplex adopts an orthogonal technique to this work and can thus be used in combination with our analysis of real-time restarts. In real-time systems, the problem of replica determinism is challenging [21]. In [17] and [8], the importance of low-cost fault-tolerance techniques without the use of redundant controllers is motivated. In [9], the notion of fault tolerance in distributed control systems is defined with respect to system operation instead of the operation of an individual controller. In this work, we adopt a similar model of fault-tolerance where we guarantee timely and correct system operation in the presence of failures (and hence restarts) of individual controllers and sensors.

Traditional techniques for failure-tolerance developed for transaction based systems [22] and scientific computing [11, 1] are not applicable to these systems primarily due to the real-time requirements and the periodic event-based semantics in this domain. Further, generic checkpointing techniques have been shown to be inadequate in [10].

7 Conclusions and Future Work

In this paper, we demonstrate techniques to maintain system operation in the presence of individual component restarts in a real-time networked control system without the use of checkpointing. We analyzed the real-time properties of a networked control system following the sensor-controller-actuator pattern, in the presence of restarts to the controller. We analyzed the effect of the state estimator in the controller for periodic state and the control buffer in the actuator. Using this analysis we derived conditions on the inter-arrival time of sensor updates and the constraint under which the periodic state is regenerated upon recovery (and hence real-time requirement of the controller satisfied) *without* checkpointing the periodic state upon each update. We tested the above analysis on our car control testbed. In order to tolerate restarts of sensors and controllers while maintaining correct and timely system operation, we implemented techniques to eliminate communication-based restart dependencies between the different components and to regenerate the aperiodic state from a checkpoint. The combination of our design and our analysis techniques allowed us to restart the car controller in our experiment with negligible deviations of the car from its trajectory.

As future work, we plan to develop a static tool to formally analyze the communication protocol between components and detect restart dependencies. In our current work, we eliminated these dependencies by design. Also, our current work assumes that the components such as the controller function in a single mode of operation. In the future, we intend to analyze multi-mode controllers where the communication and the computation modes of the controller change depending on changes in the environment and even peer restarts, thus creating new restart dependencies dynamically.

References

[1] Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *Software Engineering*, 24(2):149–159, 1998.

[2] Girish Baliga and P. R. Kumar. A middleware architecture for federated control systems. In *Proceedings of Middleware*, Rio Di Janiero, Brazil, 2003.

[3] E. F. Camacho and Carlos Bordons. *Model Predictive Control.* Springer Verlag, June 1999.

[4] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, and R. Gowda. Reducing recovery time in a small recursively restartable system. In *Proceedings of the International Conference on Dependable Systems and Networks*, Washington, D.C., June 2002.

[5] George Candea and Armando Fox. Recursive restartability: Turning the restart sledgehammer into a scalpel. In *Proc. HOTOS-VIII*, Schloss Elmau, Germany, 2001.

[6] D. Chen and M. Sanfridson. Introduction to distributed systems for real-time control. Technical Report KTH/MMK/R–98/22–SE, Mechatronics Lab, Royal Institute of Technology, KTH, Stockholm, Sweden, November 2000.

[7] C. K. Chui and G. Chen. *Kalman filtering with real-time applications.* Springer-Verlag, 1999.

[8] J.C. Cunha and M.Z. Rela. On the use of disaster prediction for failure-tolerance in feedback control systems. Washington D.C., USA, June 2002.

[9] Joao Cunha. *Low-Cost Fault Tolerance for Continuous Real-Time Control Systems.* PhD thesis, Faculdade de Ciencias e Tecnologia, Universidade de Coimbra, July 2003.

[10] P.M. Chen D.E. Lowell, S. Chandra. Exploring failure transparency and the limits of generic recovery. In *Proc. 4th USENIX Symposium on OSDI*, San Diego, CA, 2000.

[11] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[12] J. Dehn F. Cristian, B. Dancey. Fault-tolerance in air traffic control systems. *ACM Transactions on Computer Systems*, 14(3):265–286, Aug 1996.

[13] Armando Fox George Candea. Crash-only software. In *Proc. 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, HI, May 2003.

[14] Armando Fox George Candea, James Cutler. Improving availability with recursive microreboots: A soft-state system case study. 56(1-3), March 2004.

[15] S. Graham and P. R. Kumar. Time in general-purpose control systems: The control time protocol and an experimental evaluation. In *Submitted to IEEE Conference on Decision and Control*, 2004.

[16] Jim Gray. Why do computers stop and what can be done about it? In *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, 1986.

[17] M.Z. Rela J.C. Cunha, R. Maia and J.G. Silva. A study on the failure models in feedback control systems. Goteborg, Sweden, July 2001.

[18] M. Rela J.G.Silva, P.Prata and H. Madeira. Practical issues in the use of abft and a new failure model. In *Fault Toleran Computing Symposium (FTCS-28)*, pages 26–35, Munich, Germany, 1998.

[19] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *Software Engineering*, 24(2):149–159, 1998.

[20] C. R. Palmer, G. Siganos, M. Faloutsos, and P. Gibbons. The connectivity and fault-tolerance of the internet topology. In *Workshop on Network-Related Data Management (NRDM-2001)*, 2001.

[21] Stefan Poledna. *Fault Tolerant Real-time Systems.* Kluwer Academic Publishers, Nov 1995.

[22] R. Ramakrishnan and J. Gehrke. *Database Management Systems (3rd Edition).* McGraw-Hill, 2003.

[23] Lui Sha. Dependable system upgrades. In *Proceedings of IEEE Real Time System Symposium*, 1998.

[24] K. K. Toutireddy. A testbed for fault-tolerant real-time systems. Master's thesis, Univ. of Mass. at Amherst, 1996.

[25] N. Kolettis Y. Huang, C. Kintala and N. Fulton. Software rejuvenation: analysis, module and applications. In *Proc. of the 25th Int. Symposium on Fault-Tolerant Computing*, Pasadena, CA, June 1995.